TEMA 3

Encapsulación: Funciones y Acciones

Emma Rollón
erollon@cs.upc.edu

Departamento de Ciencias de la Computación

Índice

Introducción

```
#include <iostream>
using namespace std;

// Pre: dada una secuencia de naturales
// Post: escribir el factorial para cada uno de ellos
int main() {
    int x;
    while (cin >> x) {
        int f = 1;
        for (int i = 2; i <= x; ++i) {
            f = f*i;
        }
        cout << "Factorial de " << x << ": " << f << endl;
    }
}
```

factorial

Introducción: ¿Qué significa encapsular?

Encapsular
- Asociar un nombre a unas líneas de código.
- Se puede usar (llamar/invocar) ese código:
  - desde cualquier bloque del programa
  - refiriéndose únicamente al nombre
  - sin tener que reescribir las líneas asociadas
- Permite
  - Estructurar mejor el código
  - Reutilizar código
  - Comprobar una vez correctitud código
- Evita $\longrightarrow$ Repetir código casi igual

Ejemplos de tareas encapsulables:

- Calcular el factorial de un número natural.
- Calcular el máximo de dos números enteros.
- Intercambiar el valor de dos variables enteras.
- Escribir los factores primos de un número natural mayor que 0.

Introducción: Elementos clave a definir

- Especificarla
  - Precondición (Pre): situación inicial
  - Postcondición (Post): situación final
- Darle nombre.
- Identificar las variables (parámetros) relevantes. Pueden ser:
  - de entrada: su valor lo necesitamos para hacer la tarea
    $\longrightarrow$ forma parte de la Pre.
  - de salida: su valor es resultado de la tarea
    $\longrightarrow$ forma parte de la Post.
  - de entrada/salida: su valor lo necesitamos para hacer la tarea y, además, es también uno de sus resultados
    $\longrightarrow$ forma parte tanto de la Pre como de la Post.

Visualmente:

Introducción: Elementos clave en los ejemplos

- Calcular el factorial de un número natural.

Pre: $x >= 0$ (entero)
Post: f es $x!$

El valor x lo necesito para hacer razonamiento $\rightarrow$ variable de entrada
El valor f es el resultado de la tarea $\rightarrow$ variable de salida

- Calcular el máximo de dos números enteros.

Pre: x, y son enteros
Post: z es el máximo de x e y

Los valores x, y los necesito para hacer razonamiento $\rightarrow$ vars de entrada
El valor z es el resultado de la tarea $\rightarrow$ vars de salida

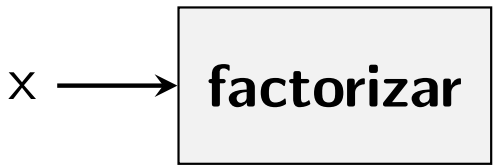
Introducción: Elementos clave en los ejemplos

- Intercambiar el valor de dos variables enteras.

Pre: x (vale A) entera, y (vale B) entera
Post: x vale B, y vale A

Las vars x, y las necesito tanto para hacer el razonamiento (entrada), y a la vez son el resultado de la tarea (salida) $\rightarrow$ vars entrada/salida.

- Escribir los factores primos de un número natural estrictamente positivo.



Pre: $x > 0$ (entero)
Post: escribe los factores primos de x.

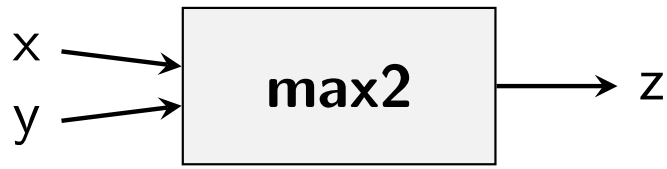
El valor x lo necesito para hacer el razonamiento $\rightarrow$ var de entrada.
El resultado de la tarea es escribir algo $\rightarrow$ NO hay variables de salida.

Funciones

Es la forma de encapsular cuando calculamos un único valor. Gralmente, identificamos que la tarea tiene:

- 1 parámetro de salida (es el valor que calculamos y devolvemos)
- y 0 o más parámetros de entrada

Ejemplos:



Funciones: cómo definirlas

Sintaxis función:

```
tipo_de_datos nombre_función(lista_parámetros_entrada) {
    I1;     // Instrucciones
    ...;
    In;
    return Expresión;  // Expresión evalúa a tipo_de_datos
}
```

Sintaxis lista_parámetros_entrada:

```
tipo_de_datos nombre_var1, tipo_de_datos nombre_var2, ...
```

Semántica:

- Los parámetros de entrada tienen un valor válido
  $\longrightarrow$ se les da valor en la llamada a (uso de) esa función.
- Se ejecutan las instrucciones I1, ..., In secuencialmente y en orden.
- Al encontrar la instrucción return se evalúa Expresión (cuyo valor será de tipo tipo_de_datos y es el resultado de la tarea) y se acaba la ejecución de la función.

Funciones: cómo definirlas

1. Calcular el factorial de un número natural.


```
// Pre: x >= 0
// Post: f es x!
int factorial(int x) {
    int f = 1;
    // Invariante: El valor de f siempre es (i-1)!
    for (int i = 2; i <= x; ++i) {
        f = f*i;
    }
    return f;
}
```

Implemento la función suponiendo que los parámetros de entrada tendrán un valor que cumpla la precondición.

Funciones: cómo definirlas

2. Calcular el máximo de dos números enteros.


```
// Pre: x (entero), y (entero)
// Post: calcula el valor máximo de x, y
int max2(int x, int y) {
    if (x >= y) return x;
    else return y;
}
```

Dado el significado de la instrucción **return**, el siguiente código es equivalente:

```
// Pre: x (entero), y (entero)
// Post: calcula el valor máximo de x, y
int max2(int x, int y) {
    if (x >= y) return x;
    return y;
}
```

Funciones: Típico error en la definición

Importante!

Todos los posibles caminos de ejecución de una función han de acabar con un return.

Un error típico es implementar una función en donde no todos los posibles caminos (flujos) de ejecución finalizan con la instrucción return.

Ejemplo:

```
int max2_incorrecto(int x, int y) {
    if (x >= y) return x;
}
```

Nos da el siguiente error de compilación:

```
Error: control reaches end of non-void function
```

Funciones: Típico error en la definición

```
int max2_incorrecto(int x, int y) {
    if (x >= y) return x;
}
```

En esta función tenemos **dos posibles caminos de ejecución**:

- Camino1: Ejecutamos el condicional, y su condición es cierta. Entonces ejecutamos instrucción return $\Rightarrow$ Correcto, su última instrucción es return.

- Camino2: Ejecutamos el condicional, y su condición es falsa. Como no hay ninguna rama else, continuamos por la siguiente instrucción que haya después del if. Como no hay ninguna, acaba la ejecución de la función $\Rightarrow$ INCORRECTO!, su última instrucción no es return.

Funciones: Típico error en la definición

Hay que tener en cuenta que:

El compilador NO analiza significado, sólo la sintaxis.

Por eso, el siguiente código también nos da el mismo error de compilación que antes:

```
int max2_incorrecto(int x, int y) {
    if (x >= y) return x;
    else if (x < y) return y;
}
```

El camino de ejecución que no acaba en return es el siguiente:

- Ejecutamos el condicional y su primera condición es falsa.
- Entonces comprobamos su segunda condición y es falsa también.
- Seguimos por la siguiente instrucción al condicional, y como no hay ninguna, acaba la ejecución de la función.

Funciones: cómo llamarlas

Sintaxis:

```
// Definición de la función: debe estar antes de su uso
tipo_de_datos nombre_funcion(param_e_1, ..., param_e_N) {
    Instrucciones;
    return Expresión;
}

int main() {
    (...)
    // Llamada a (uso de) la función
    tipo_de_datos var = nombre_funcion(E_1, ..., E_N);
    (...)
}
```

Notación:

- E_1 es una expresión del mismo tipo de datos que param_e_1.
- ...
- E_N es una expresión del mismo tipo de datos que param_e_N.

Funciones: cómo llamarlas

Semántica:

En el main, se declara la variable var, a la que se le asigna el valor que retorne la llamada a nombre_función. Para ello se siguen los siguientes pasos:

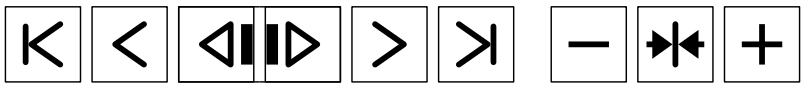
- El main espera a que nombre_función retorne un valor.
- El control de la ejecución lo toma nombre_función. Es decir, se ejecuta su código asociado:
  - Se declaran los parámetros de entrada: param_e_1 ... param_e_N
  - Se asignan: param_e_1 = E_1; ....; param_e_N = E_N
  - Se ejecuta 'Instrucciones' y se retorna la evaluación de 'Expresión'.
  - Acaba la ejecución de la función (dejan de existir param_e_1, ... param_e_N) y se devuelve el control al punto en el que se hizo la llamada (en nuestro caso al main).
- El control de la ejecución lo retoma el main, y el valor que retornó la función (es decir, Expresión) se asigna a var.

Funciones: cómo llamarlas

```
    // Pre: -
    // Post: calcula el valor máximo de x, y
    int max2(int x, int y) {
        if (x >= y) return x;
        return y;
    }

→   int main() {
        int p = 4;
        int q = 10;
        cout << max2(p,q) << endl;
    }
```

Bloque main:



Funciones: cómo llamarlas

Observación 1

El valor que retorna la función, siempre lo has de utilizar para algo.

```
int main() {
    int p = 4;
    int q = 10;
    int z = max2(p,q);              // Para asignarlo
    cout << max2(p,q) << endl; // Para escribirlo
    if (max2(p, q) > 3) {          // Como condición del if
        (...)
    }
    while (max2(p,q) < 10) {    // Como condición del bucle
        (...)
    }
    max2(p,q); // Si no utilizas su valor, para qué llamarla?
}
```

Funciones: cómo llamarlas

Observación 2

Recuerda que una expresión es una combinación de constantes, operadores, variables y funciones. En particular, puede ser una variable, lo cual evaluaría a su valor.

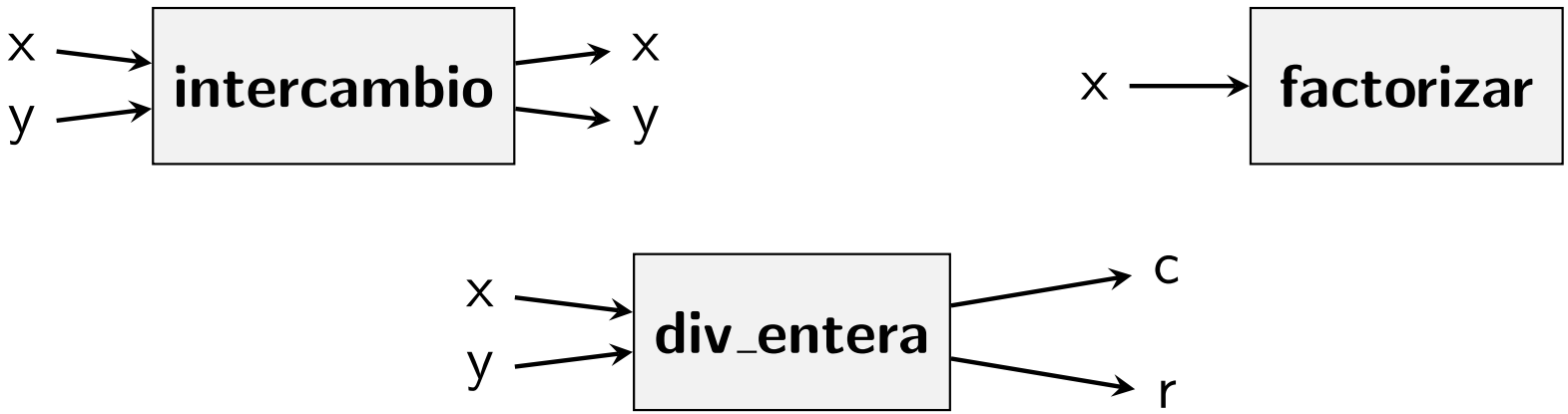
```
int main() {
    int p = 4;
    int q = 10;
    int z = max2(4,q);   // El 1r parámetro es una constante
                         // El 2o parámetro es (el valor de)
                         //   una variable
    z = max2(z + 1, p); // El 1r parámetro es combinación de:
                         //   constante, operación y variable
}
```

Acciones

Es la forma de encapsular cuando no se devuelve ningún valor (como hacen las funciones). Gralmente, identificamos que la tarea tiene:

- no tiene parámetros de salida; o,
- tiene 2 o más parámetros de salida y/o 1 o más parámetros de entrada/salida.

Ejemplos:



Acciones: cómo definirlas

Sintaxis acción:

```
void nombre_acción(lista_parámetros) {
    I1;     // Instrucciones
    ...;
    In;
}
```

Sintaxis lista_parámetros:

- parámetro de entrada (param_e):

```
tipo_de_datos nombre_var1
```

- parámetro de salida (param_s) o entrada/salida (param_e/s):

```
tipo_de_datos& nombre_var1
```

Semántica:

- Los parámetros de entrada y de entrada/salida tienen un valor válido (su valor viene dado en la llamada a (uso de) esa acción).
- Se ejecutan las instrucciones I1, ..., In secuencialmente y en orden.
- Una vez se ejecuta In, los parámetros de entrada/salida y de salida tienen el valor esperado según la Postcondición.

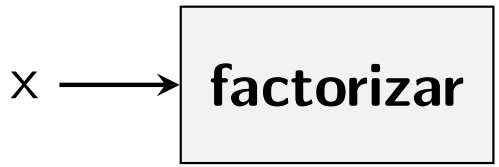
Acciones: cómo definirlas

1. Intercambiar el valor de dos variables enteras.


```
// Pre: x (vale A) entera, y (vale B) entera
// Post: x vale B, y vale A
void intercambio(int& x, int& y) {
    int aux = x;
    x = y;
    y = aux;
}
```

Acciones: cómo definirlas

2. Escribir los factores primos de un número natural estrictamente positivo.




```
// Pre: x > 0 (entero)
// Post: escribe los factores primos de x
void factorizar(int x) {
    cout << 1 << endl;
    int d = 2;
    while (x != 1) {
        if (x%d == 0) {
            cout << d << endl;
            x = x/d;
        } else ++d;
    }
}
```

Acciones: cómo definirlas

3. Dados dos naturales x e y, calcular su cociente y resto de su división entera.


```
// Pre: x, y >= 0 (enteros)
// Post: c y r son el cociente y resto de la división entera
//                de x e y, respectivamente
void div_entera(int x, int y, int& c, int& r) {
    c = x/y;
    r = x%y;
}
```

Acciones: cómo llamarlas

Sintaxis:

```
// Definición de la acción: debe estar antes de su uso
void nombre_acción(param_e, param_e/s, param_s) {
    Instrucciones;
}

int main() {
    (...)
    nombre_acción(E_param_e, var_e/s, var_s); // Llamada
    (...)
}
```

Notación:

- E_param_e es una expresión del mismo tipo de datos que el parámetro de entrada param_e.
- var_e/s es una variable del mismo tipo de datos que el parámetro de e/s param_e/s.
- var_s es una variable del mismo tipo de datos que el parámetro de salida param_s.

Acciones: cómo llamarlas

Semántica:
Al llamar a nombre_acción, el main se espera hasta que se ejecute su código asociado. Para ello se siguen los siguiente pasos:

- Se declara el parámetro de entrada param_e y se le asigna el valor E_param_e.
- param_e/s es un alias para var_e/s. Cuando param_e/s cambia de valor dentro de la acción, var_e/s sufrirá ese mismo cambio.
- param_s es un alias para var_s. Cuando param_s cambia de valor dentro de la acción, var_s sufrirá ese mismo cambio.
- Se ejecuta 'Instrucciones'.
- Acaba la ejecución de la acción (deja de existir param_e) y se devuelve el control al punto en el que se hizo la llamada (en nuestro caso al main).

Cuando el control de la ejecución vuelve al main, var_e/s y var_s tienen el valor según las asignaciones hechas en nombre_acción.

Acciones: cómo llamarlas

1. Intercambiar el valor de dos variables enteras

```
    // Pre: x (vale A) entera, y (vale B) entera
    // Post: x vale B, y vale A
    void intercambio(int& x, int& y) {
        int aux = x;
        x = y;
        y = aux;
    }

→   int main() {
        int a = 5;
        int b = 10;
        intercambio(a, b);
        cout << a << " " << b << endl;
    }
```

Bloque main:

Acciones: cómo llamarlas

2. Escribir los factores primos de un número natural estrictamente positivo.

```
// Pre: x > 0 (entero)
// Post: escribe los factores primos de x
void factorizar(int x) {
    cout << 1 << endl;
    int d = 2;
    while (x != 1) {
        if (x%d == 0) {
            cout << d << endl;
            x = x/d;
        } else ++d;
    }
}

int main() {
    int a = 15;
    factorizar(a + 5);    // escribirá los factores de 20
}
```

Acciones: cómo llamarlas

3. Dados dos naturales x e y, calcular su cociente y resto.

```
    // Pre: x, y >= 0 (enteros)
    // Post: c y r son el cociente y resto de x e y, resp.
    void div_entera(int x, int y, int& c, int& r) {
        c = x/y;
        r = x%y;
    }

→   int main() {
        int a = 15;
        int b = 10;
        int cociente, resto;  // Fíjate: NO tienen valor válido
        div_entera(a + 5, b, cociente, resto);
        cout << cociente << " " << resto << endl;
    }
```

Bloque main:

Acciones: cómo llamarlas

Observación 1

Una acción no retorna explícitamente ningún valor, por tanto, sólo se ejecutará, no hay valor a utilizar directamente como las funciones.

```
int main() {
    int a = 5;
    int b = 10;
    int z = intercambio(a, b);           // INCORRECTO!
    cout << intercambio(a, b) << endl;   // INCORRECTO!
}
```

Observación 2

Para los parámetros de salida y entrada/salida, la llamada sólo puede ser hecha con variables.

```
int main() {
    int a = 5;
    int b = 10;
    intercambio(a+1, b); // INCORRECTO! a+1 es una expresión
    intercambio(5, b);   // INCORRECTO! 5 es una expresión
}
```

Paso de parámetros

Los parámetros de una función/acción se pueden pasar de dos formas:

Paso por valor

- Es el tipo de paso cuando el parámetro es de entrada.
- Recuerda que en el momento de la llamada, se le da valor a través de una expresión (que de forma particular, puede ser el valor de una variable).
- La sintaxis es:

```
tipo_de_datos nombre_var
```

Paso por referencia

- Es el tipo de paso cuando el parámetro es de salida o entrada/salida.
- Recuerda que en el momento de la llamada, se establece un alias entre la variable con la que se llama y el nombre del parámetro en la definición.
- La sintaxis es:

```
tipo_de_datos& nombre_var
```

Paso de parámetros

Observación 1

El & importa!! Y el nombre de los parámetros NO!!

```
    // x, y: parámetros de entrada!
    void intercambio(int x, int y) {
        int aux = x;
        x = y;
        y = aux;
    }

→   int main() {
        int a = 5;
        int b = 10;
        intercambio(a, b);
        cout << a << " " << b << endl;
    }
```

main:

```
    // x, y: parámetros de entrada!
    void intercambio(int x, int y) {
        int aux = x;
        x = y;
        y = aux;
    }

→   int main() {
        int x = 5;
        int y = 10;
        intercambio(x, y);
        cout << x << " " << y << endl;
    }
```

main:

Paso de parámetros

Observación 2

Si es un parámetro de entrada/salida:

- En la definición de la función/acción:
  - La precondición nos dirá qué valores son válidos.
  - Dentro del código se usa (consulta) su valor antes de asignarle uno nuevo.
- En la llamada a la función/acción:
  - La variable con la que se hace la llamada ha de tener un valor válido.

```
// Pre: x entero (vale A), y entero (vale B)
// Post: x vale B, y vale A
void intercambio(int &x, int &y) {
    int aux = x;
    x = y;
    y = aux;
}

int main() {
    int x = 5;
    int y = 10;
    intercambio(x, y);
    cout << x << " " << y << endl;  // Escribirá 10 5
}
```

Paso de parámetros

Observación 3

Si es un parámetro SÓLO de salida:

- En la definición de la función/acción:
  - SÓLO aparece en la postcondición.
  - Dentro del código se le asigna un valor antes de consultarlo.
- En la llamada a la función/acción:
  - La variable con la que se hace la llamada NO tendrá un valor válido.

```
// Pre: n >= 0
// Post: descomposición horaria en horas (h), minutos (m),
//           segundos (s) de n
void descompon(int n, int& h, int& m, int& s) {
    // Como h, m, s son sólo de salida, en este punto NO
    // tendrán valor válido. La acción se ocupa de dárselo
    h = n/3600;
    m = (n%3600)/60;
    s = n%60;
}
```

Paso de parámetros

Observación 3

Si es un parámetro SÓLO de salida:

- En la definición de la función/acción:
  - SÓLO aparece en la postcondición.
  - Dentro del código se le asigna un valor antes de consultarlo.
- En la llamada a la función/acción:
  - La variable con la que se hace la llamada NO tendrá un valor válido.

```
// Pre: Dado una secuencia de naturales
// Post: Escribir su descomposición horaria
int main() {
    int n;
    while (cin >> n) {
        int hora, min, sec;    // No tienen valor válido
        descompon(n, hora, min, sec);
        cout << hora << " " << min << " " << sec << endl;
    }
}
```

Otros ejemplos

```
// Pre: c es una letra en minúscula
// Post: devuelve true si c es una vocal,
//                 false en caso contrario
bool es_vocal(char c) {
    return c == 'a' or c == 'e' or c == 'i' or c == 'o' or
                c == 'u';
}

// Pre: Dada un frase en letras minúsculas acabada en '.'
// Post: Escribir su número de vocales.
int main() {
    int n_vocales = 0;
    char c;
    cin >> c;
    while (c != '.') {
        if (es_vocal(c)) ++n_vocales;
        cin >> c;
    }
    cout << n_vocales << endl;
}
```

Otros ejemplos

Dado un dígito d y un natural n estictamente positivo, implementar una función que retorne cierto si n contine el dígito d, falso en caso contrario.

Con var booleana:

```
// Pre: 0 <= d <= 9, n > 0
// Post: true si n contiene d
//       false en caso contrario
bool contiene(int d, int n) {
    bool found = false;
    while (not found and n != 0) {
        if (d == n%10) found = true;
        n = n/10;
    }
    return found;
}
```

Sin var booleana:

```
// Pre: 0 <= d <= 9, n > 0
// Post: true si n contiene d
//       false en caso contrario
bool contiene(int d, int n) {
    while (n != 0) {
        if (d == n%10) return true;
        n = n/10;
    }
    return false;
}
```

Observaciones

- Las dos opciones implementan un esquema de búsqueda.
- Fíjate que si n pudiera ser 0, ninguno de los dos códigos sería correcto.

Otros ejemplos

```
// Pre: x >= 0 entero
// Post: devuelve true si x es primo,
//                 false en caso contrario
bool es_primo(int x) {
    bool prime = x > 1;
    int d = 2;
    while (prime and d*d <= x) {
        if (x %d == 0) prime = false;
        ++d;
    }
    return prime;
}
```

```
// Pre: x >= 0 entero
// Post: devuelve true si x es primo,
//                 false en caso contrario
bool es_primo(int x) {
    if (x < 2) return false;
    for (int d = 2; d*d <= x; ++d) {
        if (x %d == 0) return false;
    }
    return true;
}
```

```
// Pre: Dada una secuencia de naturales
// Post: Para cada uno de ellos escribir SI si es primo,
//          NO en caso contrario
int main() {
    int n;
    while (cin >> n) {
        if (es_primo(n)) cout << "SI" << endl;
        else cout << "NO" << endl;
    }
}
```

Otros ejemplos

```
// Pre: c es una letra
// Post: retorna c en mayúsculas si era minúscula y viceversa
char cambiar(char c) {
    if (c >= 'a' and c <= 'z') return char('A' + c - 'a');
    else return char('a' + c - 'A');
}

// Pre: Dado un natural n y n letras a continuación
// Post: Escribirlas en mayúsculas si eran minúsculas y
//           viceversa, cada una en una línea
int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        char c;
        cin >> c;
        cout << cambiar(c) << endl;
    }
}
```

Otros ejemplos

1. Escribir una función tal que dado un natural estrictamente positivo retorne la suma de sus divisores excepto él mismo.

2. Dada una secuencia de naturales estrictamente positivos, escribir para cada uno de ellos "Es perfecto"si el número es perfecto y "No es perfecto", en caso contrario. Un número es perfecto cuando su valor es igual al de la suma de sus divisores (excepto él mismo).

3. Escribir una función tal que dado un natural estrictamente positivo, retorne cierto si es un número poderoso, falso en caso contrario. Un número x es poderoso si para cada factor primo p que divide x, p^2 también lo divide.

4. Escribir una función tal que dado un natural, retorne cierto si la suma de sus dígitos en posiciones pares (empezando a contar desde su dígito de menos peso que está en posición 0) es impar, false en caso contrario.